8 – Advanced Keyboard Tricks

I often kiddingly describe Unix as “the operating system for people who like to type.” Of course, the fact that it even has a command line is a testament to that. But command line users don't like to type *that* much. Why else would so many commands have such short names like `cp`, `ls`, `mv`, and `rm`? In fact, one of the most cherished goals of the command line is laziness; doing the most work with the fewest number of keystrokes. Another goal is never having to lift our fingers from the keyboard and reach for the mouse. In this chapter, we will look at `bash` features that make keyboard use faster and more efficient.

The following commands will make an appearance:

- `clear` – Clear the screen
- `history` – Display the contents of the history list

Command Line Editing

`bash` uses a library (a shared collection of routines that different programs can use) called *Readline* to implement command line editing. We have already seen some of this. We know, for example, that the arrow keys move the cursor, but there are many more features. Think of these as additional tools that we can employ in our work. It’s not important to learn all of them, but many of them are very useful. Pick and choose as desired.

Note: Some of the key sequences below (particularly those that use the `Alt` key) may be intercepted by the GUI for other functions. All of the key sequences should work properly when using a virtual console.

Cursor Movement

The following table lists the keys used to move the cursor:

Table 8-1: Cursor Movement Commands

Key	Action

`Ctrl-a`	Move cursor to the beginning of the line.
`Ctrl-e`	Move cursor to the end of the line.
`Ctrl-f`	Move cursor forward one character; same as the right arrow key.
`Ctrl-b`	Move cursor backward one character; same as the left arrow key.
`Alt-f`	Move cursor forward one word.
`Alt-b`	Move cursor backward one word.
`Ctrl-l`	Clear the screen and move the cursor to the top-left corner. The `clear` command does the same thing.

Modifying Text

Since it's possible we might make a mistake when composing a command, we need a way to correct them efficiently. Table 8-2 describes keyboard commands that are used to edit characters on the command line.

Table 8-2: Text Editing Commands

Key	Action
`Ctrl-d`	Delete the character at the cursor location.
`Ctrl-t`	Transpose (exchange) the character at the cursor location with the one preceding it.
`Alt-t`	Transpose the word at the cursor location with the one preceding it.
`Alt-l`	Convert the characters from the cursor location to the end of the word to lowercase.
`Alt-u`	Convert the characters from the cursor location to the end of the word to uppercase.

Cutting and Pasting (Killing and Yanking) Text

The Readline documentation uses the terms *killing* and *yanking* to refer to what we would commonly call cutting and pasting. Items that are cut are stored in a buffer (a temporary storage area in memory) called the *kill-ring*.

Table 8-3: Cut and Paste Commands

Key	Action
`Ctrl-k`	Kill text from the cursor location to the end of line.
`Ctrl-u`	Kill text from the cursor location to the beginning of the line.
`Alt-d`	Kill text from the cursor location to the end of the current word.
`Alt-Backspace`	Kill text from the cursor location to the beginning of the current word. If the cursor is at the beginning of a word, kill the previous word.
`Ctrl-y`	Yank text from the kill-ring and insert it at the cursor location.

The Meta Key

If you venture into the Readline documentation, which can be found in the "READLINE" section of the `bash` man page, you will encounter the term *meta key*. On modern keyboards this maps to the `Alt` key but it wasn't always so.

Back in the dim times (before PCs but after Unix), not everybody had their own computer. What they might have had was a device called a *terminal*. A terminal was a communication device that featured a text display screen and a keyboard and just enough electronics inside to display text characters and move the cursor around. It was attached (usually by serial cable) to a larger computer or the communication network of a larger computer. There were many different brands of terminals, and they all had different keyboards and display feature sets. Since they all tended to at least understand ASCII, software developers wanting portable applications wrote to the lowest common denominator. Unix systems have an elaborate way of dealing with terminals and their different display features. Since the developers of Readline could not be sure of the presence of a dedicated extra control key, they invented one and called it *meta*. While the `Alt` key serves as the meta key on modern keyboards, you can also press and release the `Esc` key to get the same effect as holding down the `Alt` key if you're still using a terminal (which you can still do in Linux!).

Completion

Another way that the shell can help us is through a mechanism called *completion*. Completion occurs when we press the tab key while typing a command. Let's see how this

works. Given a home directory that looks like this:

```
[me@linuxbox ~]$ ls
Desktop    ls-output.txt  Pictures  Templates      Videos
Documents  Music          Public
```

Try typing the following but **don't press the `Enter` key**:

```
[me@linuxbox ~]$ ls l
```

Now press the `Tab` key.

```
[me@linuxbox ~]$ ls ls-output.txt
```

See how the shell completed the line for us? Let's try another one. Again, don't press `Enter`.

```
[me@linuxbox ~]$ ls D
```

Press `Tab`.

```
[me@linuxbox ~]$ ls D
```

No completion, just nothing. This happened because `D` matches more than one entry in the directory. For completion to be successful, the “clue” we give it has to be unambiguous. If we go further as with the following:

```
[me@linuxbox ~]$ ls Do
```

and then press `Tab`:

```
[me@linuxbox ~]$ ls Documents
```

the completion is successful.

While this example shows completion of pathnames, which is its most common use, completion will also work on variables (if the beginning of the word is a `$`), user names (if the word begins with `~`), commands (if the word is the first word on the line) and hostnames (if the beginning of the word is `@`). Hostname completion works only for hostnames listed in `/etc/hosts`.

There are a number of control and meta key sequences that are associated with completion, as listed in Table 8-4.

Table 8-4: Completion Commands

Key	Action
`Alt-?`	Display a list of possible completions. *On most systems you can also do this by pressing the* `Tab` *key a second time, which is much easier.*
`Alt-*`	Insert all possible completions. This is useful when you want to use more than one possible match.

There quite a few more that are rather obscure. A list appears in the `bash` man page under “READLINE”.

Programmable Completion

Recent versions of `bash` have a facility called *programmable completion*. Programmable completion allows you (or more likely, your distribution provider) to add additional completion rules. Usually this is done to add support for specific applications. For example, it is possible to add completions for the option list of a command or match particular file types that an application supports. Ubuntu has a fairly large set defined by default. Programmable completion is implemented by shell functions, a kind of mini shell script that we will cover in later chapters. If you are curious, try the following:

```
set | less
```

and see if you can find them. Not all distributions include them by default.

Using History

As we discovered in Chapter 1, `bash` maintains a history of commands that have been entered. This list of commands is kept in our home directory in a file called

`.bash_history`. The history facility is a useful resource for reducing the amount of typing we have to do, especially when combined with command line editing.

Searching History

At any time, we can view the contents of the history list by doing the following:

```
[me@linuxbox ~]$ history | less
```

By default, `bash` stores the last 500 commands we have entered, though most modern distributions set this value to 1000. We will see how to adjust this value in Chapter 11. Let's say we want to find the commands we used to list `/usr/bin`. This is one way we could do this:

```
[me@linuxbox ~]$ history | grep /usr/bin
```

And let's say that among our results we got a line containing an interesting command like this:

```
 88  ls -l /usr/bin > ls-output.txt
```

The `88` is the line number of the command in the history list. We could use this immediately using another type of expansion called *history expansion*. To use our discovered line, we could do this:

```
[me@linuxbox ~]$ !88
```

`bash` will expand `!88` into the contents of the 88th line in the history list. There are other forms of history expansion that we will cover in the next section.

`bash` also provides the ability to search the history list incrementally. This means we can tell `bash` to search the history list as we enter characters, with each additional character further refining our search. To start incremental search press `Ctrl-r` followed by the text we are looking for. When we find it, we can either press `Enter` to execute the command or press `Ctrl-j` to copy the line from the history list to the current command line. To find the next occurrence of the text (moving "up" the history list), press `Ctrl-r` again. To quit searching, press either `Ctrl-g` or `Ctrl-c`. Here we see it in action:

```
[me@linuxbox ~]$
```

First press `Ctrl-r`.

```
(reverse-i-search)`':
```

The prompt changes to indicate that we are performing a reverse incremental search. It is "reverse" because we are searching from "now" to some time in the past. Next, we start typing our search text. In this example, `/usr/bin`:

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

Immediately, the search returns our result. With our result, we can execute the command by pressing `Enter`, or we can copy the command to our current command line for further editing by pressing `Ctrl-j`. Let's copy it. Press `Ctrl-j`.

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Our shell prompt returns, and our command line is loaded and ready for action!

The Table 8-5 lists some of the keystrokes used to manipulate the history list.

Table 8-5: History Commands

Key	Action
`Ctrl-p`	Move to the previous history entry. This is the same action as the up arrow.
`Ctrl-n`	Move to the next history entry. This is the same action as the down arrow.
`Alt-<`	Move to the beginning (top) of the history list.
`Alt->`	Move to the end (bottom) of the history list, i.e., the current command line.
`Ctrl-r`	Reverse incremental search. This searches incrementally from the current command line up the history list.
`Alt-p`	Reverse search, nonincremental. With this key, type in the search string and press enter before the search is performed.
`Alt-n`	Forward search, nonincremental.
`Ctrl-o`	Execute the current item in the history list and advance to the next

	one. This is handy if we are trying to re-execute a sequence of commands in the history list.

History Expansion

The shell offers a specialized type of expansion for items in the history list by using the `!` character. We have already seen how the exclamation point can be followed by a number to insert an entry from the history list. There are a number of other expansion features, as described in Table 8-6.

Table 8-6: History Expansion Commands

Sequence	Action
`!!`	Repeat the last command. It is probably easier to press up arrow and enter.
`!`*`number`*	Repeat history list item *number*.
`!string`	Repeat last history list item starting with string.
`!?string`	Repeat last history list item containing string.

I would caution against using the `!string` and `!?string` forms unless you are absolutely sure of the contents of the history list items.

Many more elements are available in the history expansion mechanism, but this subject is already too arcane and our heads may explode if we continue. The HISTORY EXPANSION section of the `bash` man page goes into all the gory details. Feel free to explore!

`script`

In addition to the command history feature in `bash`, most Linux distributions include a program called `script` that can be used to record an entire shell session and store it in a file. The basic syntax of the command is as follows:

`script [`*`file`*`]`

where *file* is the name of the file used for storing the recording. If no file is specified, the file `typescript` is used. See the `script` man page for a complete list of the program's options and features.

Summing Up

In this chapter we covered *some* of the keyboard tricks that the shell provides to help hardcore typists reduce their workloads. As time goes by and we become more involved with the command line, we can refer back to this chapter to pick up more of these tricks. For now, consider them optional and potentially helpful.

Further Reading

- The Wikipedia has a good article on computer terminals: http://en.wikipedia.org/wiki/Computer_terminal

9 – Permissions

Operating systems in the Unix tradition differ from those in the MS-DOS tradition in that they are not only *multitasking* systems, but also *multi-user* systems.

What exactly does this mean? It means that more than one person can be using the computer at the same time. While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if a computer is attached to a network or the Internet, remote users can log in via `ssh` (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display. The X Window System supports this as part of its basic design.

The multiuser capability of Linux is not a recent "innovation," but rather a feature that is deeply embedded into the design of the operating system. Considering the environment in which Unix was created, this makes perfect sense. Years ago, before computers were "personal," they were large, expensive, and centralized. A typical university computer system, for example, consisted of a large central computer located in one building and terminals that were located throughout the campus, each connected to the large central computer. The computer would support many users at the same time.

To make this practical, a method had to be devised to protect the users from each other. After all, the actions of one user could not be allowed to crash the computer, nor could one user interfere with the files belonging to another user.

In this chapter we will look at this essential part of system security and introduce the following commands:

- `id` – Display user identity
- `chmod` – Change a file's mode
- `umask` – Set the default file permissions
- `su` – Run a shell as another user
- `sudo` – Execute a command as another user
- `chown` – Change a file's owner

- `chgrp` – Change a file's group ownership
- `passwd` – Change a user's password

Owners, Group Members, and Everybody Else

When we were exploring the system in Chapter 3, we may have encountered a problem when trying to examine a file such as `/etc/shadow`:

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

The reason for this error message is that, as regular users, we do not have permission to read this file.

In the Unix security model, a user may *own* files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a *group* consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Unix terms is referred to as the *world*. To find out information about your identity, use the `id` command.

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

Let's look at the output. When user accounts are created, users are assigned a number called a *user ID* (*uid*) which is then, for the sake of the humans, mapped to a username. The user is assigned a *primary group ID* (*gid*) and may belong to additional groups. The above example is from a Fedora system. On other systems, such as Ubuntu, the output may look a little different:

```
[me@linuxbox ~]$ id
uid=1000(me) gid=1000(me)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(v
ideo),46(plugdev),108(lpadmin),114(admin),1000(me)
```

As we can see, the uid and gid numbers are different. This is simply because Fedora starts its numbering of regular user accounts at 500, while Ubuntu starts at 1000. We can also

see that the Ubuntu user belongs to a lot more groups. This has to do with the way Ubuntu manages privileges for system devices and services.

So where does this information come from? Like so many things in Linux, it comes from a couple of text files. User accounts are defined in the `/etc/passwd` file and groups are defined in the `/etc/group` file. When user accounts and groups are created, these files are modified along with `/etc/shadow` which holds information about the user's password. For each user account, the `/etc/passwd` file defines the user (login) name, uid, gid, account's real name, home directory, and login shell. If we examine the contents of `/etc/passwd` and `/etc/group`, we notice that besides the regular user accounts, there are accounts for the superuser (uid 0) and various other system users.

In the next chapter, when we cover processes, we will see that some of these other "users" are, in fact, quite busy.

While many Unix-like systems assign regular users to a common group such as "users", modern Linux practice is to create a unique, single-member group with the same name as the user. This makes certain types of permission assignment easier.

Reading, Writing, and Executing

Access rights to files and directories are defined in terms of read access, write access, and execution access. If we look at the output of the `ls` command, we can get some clue as to how this is implemented:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2016-03-06 14:52 foo.txt
```

The first 10 characters of the listing are the *file attributes*. The first of these characters is the *file type*. Table 9-1 describes the file types we are most likely to see (there are other, less common types too):

Table 9-1: File Types

Attribute	File Type
`-`	A regular file.
`d`	A directory.
`l`	A symbolic link. Notice that with symbolic links, the remaining file attributes are always "rwxrwxrwx" and are dummy values. The real file attributes are those of the file the symbolic link points to.

`c`	A *character special file*. This file type refers to a device that handles data as a stream of bytes, such as a terminal or `/dev/null`.
`b`	A *block special file*. This file type refers to a device that handles data in blocks, such as a hard drive or DVD drive.

The remaining nine characters of the file attributes, called the *file mode*, represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else.

Owner	Group	World
`rwx`	`rwx`	`rwx`

Table 9-2 describes the effect the `r`, `w`, and `x` mode attributes have on files and directories:

Table 9-2: Permission Attributes

Attribute	Files	Directories
`r`	Allows a file to be opened and read.	Allows a directory's contents to be listed if the execute attribute is also set.
`w`	Allows a file to be written to or truncated, however this attribute does not allow files to be renamed or deleted. The ability to delete or rename files is determined by directory attributes.	Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set.
`x`	Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set as readable to be executed.	Allows a directory to be entered, e.g., cd *directory*.

Table 9-3 provides some examples of file attribute settings:

Table 9-3: Permission Attribute Examples

File Attributes	Meaning
`-rwx------`	A regular file that is readable, writable, and executable by the file's owner. No one else has any access.
`-rw-------`	A regular file that is readable and writable by the file's owner. No one else has any access.
`-rw-r--r--`	A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world-readable.
`-rwxr-xr-x`	A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else.
`-rw-rw----`	A regular file that is readable and writable by the file's owner and members of the file's group owner only.
`lrwxrwxrwx`	A symbolic link. All symbolic links have “dummy” permissions. The real permissions are kept with the actual file pointed to by the symbolic link.
`drwxrwx---`	A directory. The owner and the members of the owner group may enter the directory and create, rename and remove files within the directory.
`drwxr-x---`	A directory. The owner may enter the directory and create, rename, and delete files within the directory. Members of the owner group may enter the directory but cannot create, delete, or rename files.

`chmod` – Change File Mode

To change the mode (permissions) of a file or directory, the `chmod` command is used. Be aware that only the file’s owner or the superuser can change the mode of a file or directory. `chmod` supports two distinct ways of specifying mode changes: octal number representation, or symbolic representation. We will cover octal number representation first.

What the Heck is Octal?

Octal (base 8), and its cousin, *hexadecimal* (base 16) are number systems often used to express numbers on computers. We humans, owing to the fact that we (or at least most of us) were born with 10 fingers, count using a base 10 number system. Computers, on the other hand, were born with only one finger and thus do all their counting in *binary* (base 2). Their number system has only two numerals, 0 and 1. So, in binary, counting looks like this:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...

In octal, counting is done with the numerals zero through seven, like so:

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...

Hexadecimal counting uses the numerals zero through nine plus the letters “A” through “F”:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...

While we can see the sense in binary (since computers have only one finger), what are octal and hexadecimal good for? The answer has to do with human convenience. Many times, small portions of data are represented on computers as *bit patterns*. Take for example an RGB color. On most computer displays, each pixel is composed of three color components: eight bits of red, eight bits of green, and eight bits of blue. A lovely medium blue would be a 24 digit number:

010000110110111111001101

How would you like to read and write those kinds of numbers all day? I didn't think so. Here's where another number system would help. Each digit in a hexadecimal number represents four digits in binary. In octal, each digit represents three binary digits. So our 24 digit medium blue could be condensed to a six-digit hexadecimal number:

436FCD

Since the digits in the hexadecimal number “line up” with the bits in the binary number, we can see that the red component of our color is 43, the green 6F, and the blue CD.

These days, hexadecimal notation (often spoken as “hex”) is more common than octal, but as we will soon see, octal's ability to express three bits of binary will be very useful...

With octal notation, we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, this maps nicely to the

scheme used to store the file mode. Table 9-4 shows what we mean.

Table 9-4: File Modes in Binary and Octal

Octal	Binary	File Mode
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

By using three octal digits, we can set the file mode for the owner, group owner, and world.

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2016-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me   0 2016-03-06 14:52 foo.txt
```

By passing the argument “600”, we were able to set the permissions of the owner to read and write while removing all permissions from the group owner and world. Though remembering the octal to binary mapping may seem inconvenient, we will usually have only to use a few common ones: 7 (`rwx`), 6 (`rw-`), 5 (`r-x`), 4 (`r--`), and 0 (`---`).

`chmod` also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts.

- Who the change will affect
- Which operation will be performed
- What permission will be set.

To specify who is affected, a combination of the characters “u”, “g”, “o”, and “a” is used as shown in Table 9-5.

Table 9-5: `chmod` Symbolic Notation

Symbol	Meaning
`u`	Short for “user” but means the file or directory owner.
`g`	Group owner.
`o`	Short for “others” but means world.
`a`	Short for “all.” This is the combination of “u”, “g”, and “o”.

If no character is specified, “all” will be assumed. The operation may be a “+” indicating that a permission is to be added, a “-” indicating that a permission is to be taken away, or a “=” indicating that only the specified permissions are to be applied and that all others are to be removed.

Permissions are specified with the “r”, “w”, and “x” characters. Table 9-6 provides some examples of symbolic notation:

Table 9-6: `chmod` Symbolic Notation Examples

Notation	Meaning
`u+x`	Add execute permission for the owner.
`u-x`	Remove execute permission from the owner.
`+x`	Add execute permission for the owner, group, and world. This is equivalent to `a+x`.
`o-rw`	Remove the read and write permissions from anyone besides the owner and group owner.
`go=rw`	Set the group owner and anyone besides the owner to have read and write permission. If either the group owner or the world previously had execute permission, it is removed.
`u+x,go=rx`	Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

Some people prefer to use octal notation, and some folks really like the symbolic. Symbolic notation does offer the advantage of allowing us to set a single attribute without disturbing any of the others.

Take a look at the `chmod` man page for more details and a list of options. A word of caution regarding the “--recursive” option: it acts on both files and directories, so it's not as

useful as we would hope since we rarely want files and directories to have the same permissions.

Setting File Mode with the GUI

Now that we have seen how the permissions on files and directories are set, we can better understand the permission dialogs in the GUI. In both Files (GNOME) and Dolphin (KDE), right-clicking a file or directory icon will expose a properties dialog. Here is an example from GNOME:

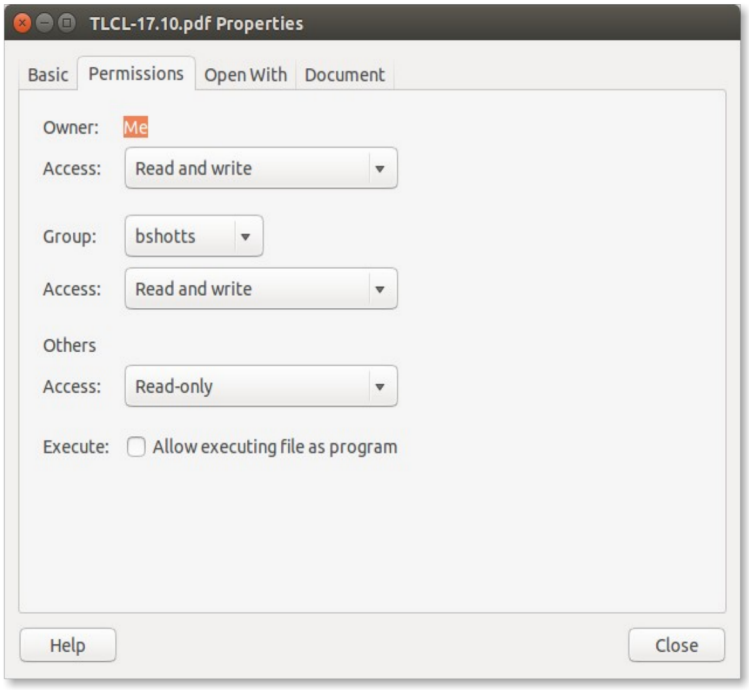

Figure 2: GNOME file permissions dialog

Here we can see the settings for the owner, group, and world.

`umask` – Set Default Permissions

The `umask` command controls the default permissions given to a file when it is created. It uses octal notation to express a *mask* of bits to be removed from a file's mode attributes. Let's take a look.

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2018-03-06 14:53 foo.txt
```

We first removed any old copy of `foo.txt` to make sure we were starting fresh. Next, we ran the `umask` command without an argument to see the current value. It responded with the value `0002` (the value `0022` is another common default value), which is the octal representation of our mask. We next create a new instance of the file `foo.txt` and observe its permissions.

We can see that both the owner and group get read and write permission, while everyone else only gets read permission. The reason that world does not have write permission is because of the value of the mask. Let's repeat our example, this time setting the mask ourselves.

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me    me   0 2018-03-06 14:58 foo.txt
```

When we set the mask to `0000` (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we take the mask, expand it into binary, and then compare it to the attributes we can see what happens.

Original file mode	--- rw- rw- rw-
Mask	000 000 000 010
Result	--- rw- rw- r--

Ignore for the moment the leading zeros (we'll get to those in a minute) and observe that where the 1 appears in our mask, an attribute was removed—in this case, the world write permission. That's what the mask does. Everywhere a 1 appears in the binary value of the mask, an attribute is unset. If we look at a mask value of `0022`, we can see what it does.

Original file mode	--- rw- rw- rw-
Mask	000 000 010 010
Result	--- rw- r-- r--

Again, where a 1 appears in the binary value, the corresponding attribute is unset. Play with some values (try some sevens) to get used to how this works. When you're done, remember to clean up.

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

Most of the time we won't have to change the mask; the default provided by the distribution will be fine. In some high-security situations, however, we will want to control it.

Some Special Permissions

Though we usually see an octal permission mask expressed as a three-digit number, it is more technically correct to express it in four digits. Why? Because, in addition to read, write, and execute permission, there are some other, less used, permission settings.

The first of these is the *setuid bit* (octal 4000). When applied to an executable file, it sets the *effective user ID* from that of the real user (the user actually running the program) to that of the program's owner. Most often this is given to a few programs owned by the superuser. When an ordinary user runs a program that is "*setuid root*" , the program runs with the effective privileges of the superuser. This allows the program to access files and directories that an ordinary user would normally be prohibited from accessing. Clearly, because this raises security concerns, the number of setuid programs must be held to an absolute minimum.

The second less-used setting is the *setgid bit* (octal 2000), which, like the setuid bit, changes the *effective group ID* from the *real group ID* of the real user to that of the file owner. If the setgid bit is set on a directory, newly created files in the directory will be given the group ownership of the directory rather the group ownership of the file's creator. This is useful in a shared directory when members of a common group need access to all the files in the directory, regardless of the file owner's primary group.

The third is called the *sticky bit* (octal 1000). This is a holdover from ancient Unix, where it was possible to mark an executable file as "not swappable." On

files, Linux ignores the sticky bit, but if applied to a directory, it prevents users from deleting or renaming files unless the user is either the owner of the directory, the owner of the file, or the superuser. This is often used to control access to a shared directory, such as `/tmp`.

Here are some examples of using `chmod` with symbolic notation to set these special permissions. Here's an example of assigning setuid to a program:

```
chmod u+s program
```

Next, here's and example of assigning setgid to a directory:

```
chmod g+s dir
```

Finally, here's an example of assigning the sticky bit to a directory:

```
chmod +t dir
```

When viewing the output from `ls`, you can determine the special permissions. Here are some examples. First, an example of a program that is setuid:

```
-rwsr-xr-x
```

Here's an example of a directory that has the setgid attribute:

```
drwxrwsr-x
```

Here's an example of a directory with the sticky bit set:

```
drwxrwxrwt
```

Changing Identities

At various times, we may find it necessary to take on the identity of another user. Often we want to gain superuser privileges to carry out some administrative task, but it is also possible to "become" another regular user for such things as testing an account. There are three ways to take on an alternate identity.

1. Log out and log back in as the alternate user.
2. Use the `su` command.
3. Use the `sudo` command.

We will skip the first technique since we know how to do it and it lacks the convenience of the other two. From within our own shell session, the `su` command allows us to assume the identity of another user and either start a new shell session with that user's ID, or to issue a single command as that user. The `sudo` command allows an administrator to set up a configuration file called `/etc/sudoers` and define specific commands that

particular users are permitted to execute under an assumed identity. The choice of which command to use is largely determined by which Linux distribution you use. Your distribution probably includes both commands, but its configuration will favor either one or the other. We'll start with `su`.

su – Run a Shell with Substitute User and Group IDs

The `su` command is used to start a shell as another user. The command syntax looks like this:

```
su [-[l]] [user]
```

If the "-l" option is included, the resulting shell session is a *login shell* for the specified user. This means the user's environment is loaded and the working directory is changed to the user's home directory. This is usually what we want. If the user is not specified, the superuser is assumed. Notice that (strangely) the `-l` may be abbreviated as `-`, which is how it is most often used. To start a shell for the superuser, we would do this:

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]#
```

After entering the command, we are prompted for the superuser's password. If it is successfully entered, a new shell prompt appears indicating that this shell has superuser privileges (the trailing # rather than a `$`), and the current working directory is now the home directory for the superuser (normally `/root`). Once in the new shell, we can carry out commands as the superuser. When finished, enter `exit` to return to the previous shell.

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

It is also possible to execute a single command rather than starting a new interactive command by using `su` this way.

```
su -c 'command'
```

Using this form, a single command line is passed to the new shell for execution. It is im-

portant to enclose the command in quotes, as we do not want expansion to occur in our shell, but rather in the new shell.

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Password:
-rw------- 1 root root     754 2007-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
total 0
[me@linuxbox ~]$
```

sudo – Execute a Command as Another User

The sudo command is like su in many ways but has some important additional capabilities. The administrator can configure sudo to allow an ordinary user to execute commands as a different user (usually the superuser) in a controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another important difference is that the use of sudo does not require access to the superuser's password. To authenticating using sudo, requires the user's own password. Let's say, for example, that sudo has been configured to allow us to run a fictitious backup program called "backup_script", which requires superuser privileges. With sudo it would be done like this:

```
[me@linuxbox ~]$ sudo backup_script
Password:
System Backup Starting...
```

After entering the command, we are prompted for our password (not the superuser's) and once the authentication is complete, the specified command is carried out. One important difference between su and sudo is that sudo does not start a new shell, nor does it load another user's environment. This means that commands do not need to be quoted any differently than they would be without using sudo. Note that this behavior can be overridden by specifying various options. Note, too, that sudo can be used to start an interactive superuser session (much like su -) by using the -i option. See the sudo man page for details.

To see what privileges are granted by sudo, use the -l option to list them:

```
[me@linuxbox ~]$ sudo -l
```

```
User me may run the following commands on this host:
    (ALL) ALL
```

Ubuntu and sudo

One of the recurrent problems for regular users is how to perform certain tasks that require superuser privileges. These tasks include installing and updating software, editing system configuration files, and accessing devices. In the Windows world, this is often done by giving users administrative privileges. This allows users to perform these tasks. However, it also enables programs executed by the user to have the same abilities. This is desirable in most cases, but it also permits *malware* (malicious software) such as viruses to have free rein of the computer.

In the Unix world, there has always been a larger division between regular users and administrators, owing to the multiuser heritage of Unix. The approach taken in Unix is to grant superuser privileges only when needed. To do this, the `su` and `sudo` commands are commonly used.

Up until a few of years ago, most Linux distributions relied on `su` for this purpose. `su` didn't require the configuration that `sudo` required, and having a root account is traditional in Unix. This introduced a problem. Users were tempted to operate as root unnecessarily. In fact, some users operated their systems as the root user exclusively, since it does away with all those annoying “permission denied” messages. This is how you reduce the security of a Linux system to that of a Windows system. Not a good idea.

When Ubuntu was introduced, its creators took a different tack. By default, Ubuntu disables logins to the root account (by failing to set a password for the account) and instead uses `sudo` to grant superuser privileges. The initial user account is granted full access to superuser privileges via `sudo` and may grant similar powers to subsequent user accounts.

`chown` – Change File Owner and Group

The `chown` command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of `chown` looks like this:

```
chown [owner][:[group]] file...
```

`chown` can change the file owner and/or the file group owner depending on the first argument of the command. Table 9-7 provides some examples.

Table 9-7: `chown` Argument Examples

Argument	Results
`bob`	Changes the ownership of the file from its current owner to user `bob`.
`bob:users`	Changes the ownership of the file from its current owner to user `bob` and changes the file group owner to group `users`.
`:admins`	Changes the group owner to the group `admins`. The file owner is unchanged.
`bob:`	Changes the file owner from the current owner to user `bob` and changes the group owner to the login group of user `bob`.

Let's say we have two users; `janet`, who has access to superuser privileges and `tony`, who does not. User `janet` wants to copy a file from her home directory to the home directory of user `tony`. Since user `janet` wants `tony` to be able to edit the file, `janet` changes the ownership of the copied file from `janet` to `tony`.

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 root  root  root 2018-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 tony  tony  tony 2018-03-20 14:30 /home/tony/myfile.txt
```

Here we see user `janet` copy the file from her directory to the home directory of user `tony`. Next, `janet` changes the ownership of the file from `root` (a result of using `sudo`) to `tony`. Using the trailing colon in the first argument, `janet` also changed the group ownership of the file to the login group of `tony`, which happens to be group `tony`.

Notice that after the first use of `sudo`, `janet` was not prompted for her password. This is because `sudo`, in most configurations, “trusts” us for several minutes until its timer

runs out.

`chgrp` – Change Group Ownership

In older versions of Unix, the `chown` command only changed file ownership, not group ownership. For that purpose, a separate command, `chgrp` was used. It works much the same way as `chown`, except for being more limited.

Exercising Our Privileges

Now that we have learned how this permissions thing works, it's time to show it off. We are going to demonstrate the solution to a common problem—setting up a shared directory. Let's imagine that we have two users named “bill” and “karen.” They both have music collections and want to set up a shared directory, where they will each store their music files as Ogg Vorbis or MP3. User `bill` has access to superuser privileges via `sudo`.

The first thing that needs to happen is a group needs to be created that will have both `bill` and `karen` as members. Using the graphical user management tool, `bill` creates a group called `music` and adds users `bill` and `karen` to it:

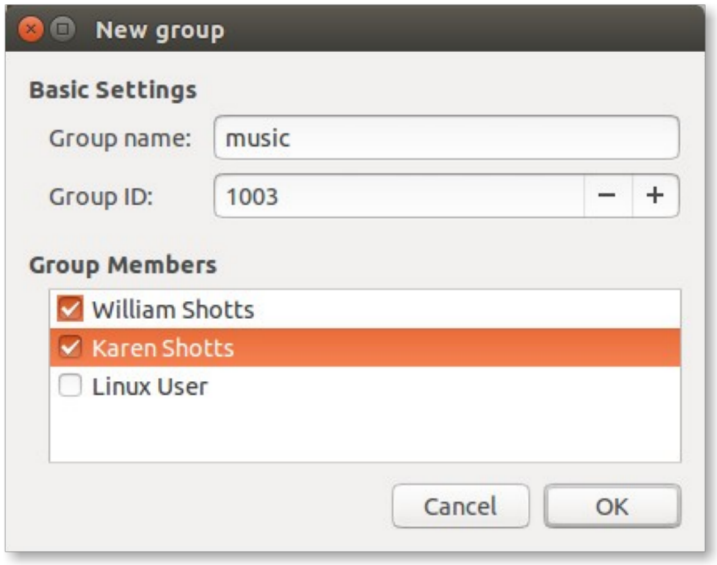

Figure 3: Creating a new group with GNOME

Next, `bill` creates the directory for the music files.

```
[bill@linuxbox ~]$ sudo mkdir /usr/local/share/Music
Password:
```

Since `bill` is manipulating files outside his home directory, superuser privileges are required. After the directory is created, it has the following ownerships and permissions:

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2018-03-21 18:05 /usr/local/share/Music
```

As we can see, the directory is owned by `root` and has permission mode 755. To make this directory sharable, `bill` needs to change the group ownership and the group permissions to allow writing.

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2018-03-21 18:05 /usr/local/share/Music
```

What does this all mean? It means that we now have a directory, `/usr/local/share/Music` that is owned by `root` and allows read and write access to group `music`. Group `music` has members `bill` and `karen`; thus, `bill` and `karen` can create files in directory `/usr/local/share/Music`. Other users can list the contents of the directory but cannot create files there.

But we still have a problem. With the current permissions, files and directories created within the `Music` directory will have the normal permissions of the users `bill` and `karen`.

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill   bill   0 2018-03-24 20:03 test_file
```

Actually there are two problems. First, the default `umask` on this system is 0022, which prevents group members from writing files belonging to other members of the group. This would not be a problem if the shared directory contained only files, but since this directory will store music, and music is usually organized in a hierarchy of artists and albums, members of the group will need the ability to create files and directories inside directories created by other members. We need to change the `umask` used by `bill` and

`karen` to 0002 instead.

Second, each file and directory created by one member will be set to the primary group of the user rather than the group `music`. This can be fixed by setting the setgid bit on the directory.

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2018-03-24 20:03 /usr/local/share/Music
```

Now we test to see whether the new permissions fix the problem. `bill` sets his `umask` to 0002, removes the previous test file, and creates a new test file and directory:

```
[bill@linuxbox ~]$ umask 0002
[bill@linuxbox ~]$ rm /usr/local/share/Music/test_file
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 bill   music 4096 2018-03-24 20:24 test_dir
-rw-rw-r-- 1 bill   music    0 2018-03-24 20:22 test_file
[bill@linuxbox ~]$
```

Both files and directories are now created with the correct permissions to allow all members of the group `music` to create files and directories inside the `Music` directory.

The one remaining issue is `umask`. The necessary setting only lasts until the end of session and must be reset. In Chapter 11, we'll look at making the change to `umask` permanent.

Changing Your Password

The last topic we'll cover in this chapter is setting passwords for yourself (and for other users if you have access to superuser privileges). To set or change a password, the `passwd` command is used. The command syntax looks like this:

```
passwd [user]
```

To change your password, just enter the `passwd` command. You will be prompted for your old password and your new password.

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
```

The `passwd` command will try to enforce use of “strong” passwords. This means it will refuse to accept passwords that are too short, are too similar to previous passwords, are dictionary words, or are too easily guessed.

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
BAD PASSWORD: is too similar to the old one
New UNIX password:
BAD PASSWORD: it is WAY too short
New UNIX password:
BAD PASSWORD: it is based on a dictionary word
```

If you have superuser privileges, you can specify a username as an argument to the `passwd` command to set the password for another user. Other options are available to the superuser to allow account locking, password expiration, and so on. See the `passwd` man page for details.

Summing Up

In this chapter we saw how Unix-like systems such as Linux manage user permissions to allow the read, write, and execution access to files and directories. The basic ideas of this system of permissions date back to the early days of Unix and have stood up pretty well to the test of time. But the native permissions mechanism in Unix-like systems lacks the fine granularity of more modern systems.

Further Reading

- Wikipedia has a good article on malware: http://en.wikipedia.org/wiki/Malware

There are number of command line programs used to create and maintain users and groups. For more information, see the man pages for the following commands:

- `adduser`
- `useradd`
- `groupadd`